# 1 INTRODUCTION

*Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.*

— MR. Cox ***1893**, Cinderella: Three hundred and forty-five variants [1]*

THE first web browser, Nexus, made its appearance in 1990 [2]. At its inception, web browsing consisted solely of retrieving and displaying small, static text pages. With Nexus, users could access for the first time interlinked hypertext documents, so-called HTML pages. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet-based companies, and the demand for more engaging user experiences gave rise the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [3], a programming language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers offer JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Consequently, over the past several decades, web browsers have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers.

Despite being the main scripting language in modern web browsers, JavaScript possesses inherent limitations due to its unique language characteristics [4]. Each JavaScript engine requires the parsing and recompiling of the JavaScript code, thereby causing substantial overhead. In practice, the process of parsing and compiling JavaScript code constitutes the majority of website load times [1]. Additionally, JavaScript presents security issues, including the lack of memory

[0] Compilation probe time 2023/11/21 13:05:37

[1] `https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast`

isolation, which potentially enables information extraction from other processes [5, 6]. Numerous attempts have been made to port other languages, offering different guarantees, to the browser execution as alternatives to JavaScript. For instance, Java applets emerged on web pages in the late 90s, enabling the execution of Java bytecode on the client side [2]. Likewise, Microsoft attempted twice with ActiveX in 1996 [3], and Silverlight in 2007 [4]. However, these attempts either failed to persist or experienced low adoption, primarily due to security issues and the absence of consensus among the community of browser vendors.

Importantly, in 2014, Alon Zakai and colleagues proposed the Emscripten tool [5]. Emscripten employs a strict subset of JavaScript, asm.js, to facilitate the compilation of low-level code such as C to JavaScript. Asm.js was included as an LLVM backend [6]. This strategy offered the advantages of the ahead-of-time optimizations from LLVM, resulting in performance gains on browser clients [7] when compared to standard JavaScript code. Asm.js outperformed JavaScript because it restricted language features to those that could be optimized in the LLVM pipeline. Moreover, it eliminated most of the language's dynamic characteristics, limiting it to numerical types, top-level functions, and one large array in memory accessed directly as raw data. Asm.js proved that client-side code could be enhanced with the appropriate language design and standardization. In response to persistent JavaScript-related issues, the formalization and creation of a formal specification following asm.js laid the groundwork for the emergence of WebAssembly. In 2015, the Web Consortium (W3C) standardized WebAssembly as a bytecode for the web environment. As a result, WebAssembly bytecode became the fourth official language for the web.

The first distinction from earlier attempts to port non-JavaScript languages to the web lies in WebAssembly's initial design. Unlike its predecessors, WebAssembly was crafted to supplement JavaScript in the browser as a platform-agnostic, low-level bytecode, rather than to completely replace it. Its primary goal was to replace computing-intensive JavaScript code in contemporary web applications. Additionally, WebAssembly is the inaugural major language that utilized formal specification and verification right from the design inception [7, 8].

Importantly, WebAssembly provides a platform for compiling several legacy code applications, like those written in C/C++. For example, LLVM includes WebAssembly as a backend since release 7.1.0 published in May 2019[8]. Therefore, the emergence of WebAssembly, a fast, low-level, portable bytecode for browsers, has the potential to transform web software as we know it. It paves the way

[2] https://www.oracle.com/java/technologies/javase/9-deprecated-features.html

[3] https://web.archive.org/web/20090828024117/http://www.microsoft.com/presspass/press/1996/mar96/activxpr.mspx

[4] https://www.microsoft.com/silverlight/

[5] https://emscripten.org/

[6] http://asmjs.org/spec/latest/

[7] ttps://hacks.mozilla.org/2015/03/asm-speedups-everywhere/

[8] https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0

for web applications to undertake roles traditionally reserved for native desktop applications. For example, applications such as AutoCAD and Adobe Photoshop have been ported to WebAssembly[9].

The WebAssembly specification embodies several language design principles that pave the way for its extension beyond the web ecosystem. For instance, the architecture of WebAssembly guarantees self-containment. Inherently, WebAssembly binaries are prohibited from accessing memory beyond their own designated space, thereby amplifying security via Software Fault Isolation (SFI) policies [9]. Consequently, research has highlighted the benefits of integrating WebAssembly as an intermediate layer in contemporary cloud platforms [10]. In particular, the employment of WebAssembly binaries improves startup times and optimizes memory consumption, outperforming virtualization and containerization [11]. Furthermore, compared to virtual machines and containers, WebAssembly programs are more compact, highlighting their efficient deployment, especially when network transportation is a consideration. The methodology for standalone WebAssembly execution was formalized in 2019 when the Bytecode Alliance proposed the WebAssembly System Interface (WASI) [10]. WASI standardizes the execution of WebAssembly via a POSIX-like interface protocol, thereby facilitating the execution of WebAssembly closer operating system. This standardization enables WebAssembly to function outside web browsers, extending its use to cloud environments and IoT devices.

The extensive applicability and rapid adoption of WebAssembly have prompted requests for additional features. However, these demands do not always align with the initial specifications. For extending WebAssembly with a new proposal, it must satisfy particular criteria. A new proposal needs a formal specification and a minimum of two independent implementations, e.g., two different WebAssembly engines. This approach allows for swift incorporation of new formalization and features via the so-called "evergreen method", while maintaining the original WebAssembly specification intact. Since the inception of WebAssembly, numerous extensions have been proposed for standardization. For instance, the SIMD proposal enables the execution of vectorized instructions in WebAssembly. After approval, new extensions remain optional, ensuring that the core WebAssembly version remains 1.0. The ongoing development of WebAssembly provides avenues for research and development. However, it also gives rise to security concerns within the ecosystem, as new threats emerge.

## 1.1 Predictability in WebAssembly ecosystems

Over the past three decades, web browsers and JavaScript have significant evolution, leading to myriad implementations. However, only Firefox, Chrome,

[9] https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw

[10] https://github.com/WebAssembly/WASI

Safari, and Edge are typically utilized on devices. Web page resources, including those containing WebAssembly binaries, are primarily served from centralized datacenters [12]. This situation creates a highly predictable ecosystem, where potential attackers can predict ecosystem behavior, from the browser to the code it executes. This predictability may be exploited to launch large-scale attacks, as predictability inherently increases the chances of successful attacks [13]. For example, if one-quarter of all devices operate the same code in the same browser, a single flaw could impact millions of devices in the same way [14].

The aforementioned issue is exacerbated when considering the adoption of WebAssembly by edge-cloud computing platforms to provide services. In addition to browser clients, thousands of edge devices operate millions of identical WebAssembly instantiations per second. This suggests that a single vulnerable WebAssembly binary in an edge network node could render every node identically susceptible due to the binary replication occurring on each node. A potential attacker could compromise all edge nodes concurrently, implying that a single distributed WebAssembly binary could trigger a global attack [11].

We devise two scenarios where predictability affects WebAssembly ecosystems. First, the predictability of execution engines and WebAssembly binaries themselves facilitates side-channel and memory attacks. Despite the praise for WebAssembly's security, particularly its design that prohibits programs from accessing data beyond their own memory, it is not immune to such vulnerabilities. For example, Rokicki et al. highlighted the potential risk of port contention side-channel attacks using WebAssembly malware in browsers [15]. In such cases, mitigations often involve hardware and operating-level changes, which are not always feasible. Moreover, attacks within the memory of WebAssembly itself are feasible[16, 17] as innate vulnerabilities can exist in WebAssembly binaries due to flaws in the source code. Besides, the lack of stack-smashing protections could result in unnoticed overflows and crashes during WebAssembly executions[18]. In standalone deployments, Genkin et al. demonstrated the possibility of data extraction via cache-timing side channels in WebAssembly [19]. In a similar vein, Maisuradze and Rossow exhibited speculative execution attacks on WebAssembly binaries [20].

Second, the defenses for identifying and addressing vulnerabilities are generally predictable. In particular, this predictability can be manipulated by malicious actors to create programs aimed at deceiving these defense mechanisms. For example, malware can be distributed via WebAssembly binaries. The capability of WebAssembly for efficient computation makes it an appealing target for misuse by cybercriminals, especially for cryptojacking [21]. The challenge in identifying and eliminating cryptojacking enables it to function persistently on a victim's computer, constantly utilizing resources and generating income for the attacker [22]. Several techniques, such as static analysis, dynamic analysis,

[11] `https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem`

and even sophisticated machine learning methods, are successfully applied to detect WebAssembly malware [23, 24, 25, 26, 27, 28]. However, most of these research works do not consider the predictability of an attacker knowing that a WebAssembly program is not treated as obfuscated.

## 1.2 Problems statements

To sum up, predictability and potential vulnerabilities form a harmful combination. This principle does not exclude WebAssembly and its ecosystem. The effect of exploiting a single vulnerability in WebAssembly could prove catastrophic, given all devices running the same WebAssembly binaries could be affected. On the other hand, WebAssembly malware pose a severe threat. Present defenses may not adequately protect against them, as they have not been designed to manage situations outside predictable scenarios, such as obfuscation. Besides, mitigations might require hardware and operating-level changes, which are not always feasible. In this dissertation, we tackle the subsequent two problems:

**P1 The WebAssembly ecosystem and binaries are susceptible to attacks, especially those from side-channel threats.**

**P2 WebAssembly malware presents a substantial threat. Predictability leads to the assumption that malware is typically considered unique.**

## 1.3 Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problems via Software Diversification. Software Diversification is a security strategy that involves identifying, developing, and deploying program variants of a given original program [29]. Pioneers in this field, Cohen et al. [30] and Forrest et al. [31], proposed enhancing software diversity through code transformations. Their proposal recommended the creation of diverse program variants, maintaining their original functionalities. The aim of Software Diversification is to lessen potential vulnerabilities by enhancing their behaviour unpredictability.

Studies have demonstrated that Software Diversification effectively removes vulnerabilities. Eichin et al., in work in 1989, underscored the practical benefits of diversification [32]. They illustrated how diversification limited the exploitation of the Morris Worm to a few machines. From an attacker's perspective, the diversity of target systems rendered them unpredictable. For WebAssembly, Software Diversification could bolster browsers and standalone engines by providing diversified program variants, thereby making it harder for attackers to exploit vulnerabilities, addressing **P1**. Furthermore, it could increase the accuracy of WebAssembly malware detectors and WebAssembly

| Contribution | Research papers | | | |
|---|---|---|---|---|
| | I [33] | II [34] | III [35] | IV [36] |
| C1 Defensive diversification | ✓ | ✓ | ✓ | |
| C2 Offensive diversification | | | | ✓ |
| C3 Experimental contribution | ✓ | ✓ | ✓ | ✓ |
| C4 Theoretical contribution | ✓ | | ✓ | |

***Table 1.1:*** *Mapping between contributions and research papers.*

analysis tools in general, addressing **P2**. However, the implementation of Software Diversification in WebAssembly is still largely unexplored. In light of this, we offer the following contributions within the context of Software Diversification, which are not necessarily mutually exclusive.

**C1 Defensive Diversification:** In order to address **P1**, we assess how diversified WebAssembly program variants could be used for defensive purposes. We provide empirical insights about the practical usage of the generated variants in preventing attacks.

**C2 Offensive Diversification:** In order to address **P2**, we evaluate the potential for using generated WebAssembly program variants for offensive purposes. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

**C3 Experimental contribution:** For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research on **P1** and **P2**.

**C4 Theoretical contribution:** We propose a theoretical foundation in order to generate and improve Software Diversification for WebAssembly. We provide a formal definition of WebAssembly program variants and their diversity. We also provide a formal definition of WebAssembly program diversity generation.

## 1.4 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

**I**: **CROW: Code randomization for WebAssembly bytecode.**
**Javier Cabrera-Arteaga**, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
*Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages*
`https://doi.org/10.14722/madweb.2021.23004`

**Summary:** In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements Software Diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create thousands of statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

**II**: **Multivariant execution at the Edge.**
**Javier Cabrera-Arteaga**, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
*Moving Target Defense (MTD 2022), 12 pages*
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

**Summary:** In this paper, we synthesize functionally equivalent variants of deployed edge services. Service variants are encapsulated into a single multivariant WebAssembly binary. A random variant is selected and executed each time a function is invoked. Execution of multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

**III**: **Wasm-mutate: Fast and efficient Software Diversification for WebAssembly.**
**Javier Cabrera-Arteaga**, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
*Submitted to Computers & Security, under revision, 17 pages*
`https://arxiv.org/pdf/2309.07638.pdf`

**Summary:** This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly generate functionally equivalent yet behaviorally diverse WebAssembly variants by randomly traversing e-graphs. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

**IV**: **WebAssembly Diversification for Malware evasion.**
**Javier Cabrera-Arteaga**, Tim Toady, Martin Monperrus, Benoit Baudry
*Computers & Security, Volume 131, 2023, 17 pages*

**Summary:** WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiming WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

## ■ Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [37]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.